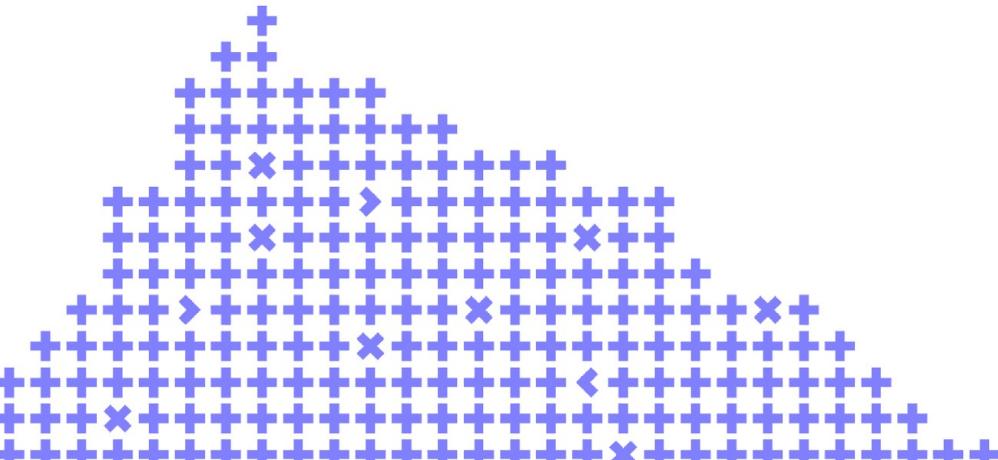


# How we cook FoundationDB

Daniil Gitelson



Co-organizer

**Yandex**

# Introduction slide

In Lekton we develop payment processing system for Banks and Wallets

Our biggest installation > 200M Cards

Oracle-based solution

- Pricy
- Not easy to scale

Customers asked what we could do about it



# What we need from DBMS

- ACID (Multi-entry transactions)
- Distributed
- HA, Geo Replication
- Administration simplicity
- Throughput over latency (not counting milliseconds)

# FoundationDB

Open Source

Distributed transactional ordered KV store

Consistency over availability (CP)

Crazy on correctness: special framework for simulating failures



# How FoundationDB checks the boxes

- Distributed ✓
- ACID ✓
  - Serializable isolation level
  - Distributed transactions
  - Optimistic locking
- HA, Geo Replication ✓
- Administration simplicity ✓
- Throughput over latency, ✓  
automatic batching, >6ms for any write

# FoundationDB: What we've built on top

- Document-like storage
  - Indexes
  - Partitioning
- Sequences
- Pessimistic locks
- Transactional outbox
- Encryption



# FoundationDB: What we've built on top

- **Document-like storage**

- **Indexes**
- **Partitioning**

- Sequences

- Pessimistic locks

- **Transactional outbox**

- Encryption

# FoundationDB Data Model

Ordered byte array => byte array, infinite space  
stored as index organized table

Operations:

- GET, GETRANGE
- SET
- CLEAR
- CLEARRANGE
- ???

# How to module document-like storage

Storing as pair: ID => Serialized document

1234 => {"username": "daniil"}

Actual value can be stored in JSON, Protobuf, etc, we use  
CBOR: it's compact enough + self descriptive which is  
easier to troubleshoot

Storing different object types => need to have prefix:  
user:1234 => {"username": "daniil"}

# How do we model unique index

User: {"ID": 123, "username": "daniil"}, unique index by username

**SET** user:123, {"username": "daniil"}

**SET** username:daniil, 123

**The bad:** looking up by index takes 2 reads:

**The good:** query scales with nodes (unlike e.g. in Mongo)

It's good practice to use 'business' PK to optimize common lookup scenario



How do we model unique index: Uniqueness check

**SET** does not check whether key exists or not

Naive check: under **REPEATABLE READ** isolation level

1. Start transaction
2. **GET** daniil
3. **SET** daniil => 123
4. **SET** 123 => {"ID": 123, "username": "daniil"}
5. Commit

How do we model unique index: Uniqueness check

**SET** does not check whether key exists or not

Naive check: under **REPEATABLE READ** isolation level

1. Start transaction
2. **GET daniil: No lock acquired, nothing to lock on**
3. **SET daniil => 123**
4. **SET 123 => {"ID": 123, "username":"daniil"}**
5. Commit

How do we model unique index: Uniqueness check

**SERIALIZABLE** works as ‘Lock every query result’

1. Start transaction
2. **GET** daniil
3. **SET** daniil => 123
4. **SET** 123 => {"ID": 123, "username":"daniil"}
5. Commit

How do we model unique index: Uniqueness check

**SERIALIZABLE** works as ‘Lock every query result’

1. Start transaction
2. **GET** daniil: Blocks other transactions affecting key ‘daniil’
3. **SET** daniil => 123
4. **SET** 123 => {"ID": 123, "username":"daniil"}
5. Commit

How do we model non unique index: First try

Lets reduce the problem to the previous:  
secondary key => [array, of, primary, keys]

- Good for high-cardinality indexes (few values have same index)
- Less efficient as the list grows
  - Contention when adding multiple values with same secondary key
  - Bigger updates for adding single value

How do we model non unique index: Second try

Secondary key + separator + primary key => empty value

daniil:1234 => empty value

daniil:1235 => empty value

Query: Find all by key prefix + resolve by primary keys

No contention on insert

Update doesn't grow

Slightly more space used  
due to index value  
replication

# Payments history model

Payment is some arbitrary data with given client id and a timestamp

Most typical query- get payments for client for the last month (ordered by timestamp)

Requirements:

- Efficient query
- Efficient removal until given month

# Payments history model: First try

Key: CLIENT\_ID:YYYYMMDD\_HHMMSS

Get history for the client for current month:

GETRANGE with prefix: 1234:202212

Index organized table =>

Transactions are layed out together on disk => less IOPS

No way to easily remove all data before given date :(

No partitions in FoundationDB => we need to fake them

# Payments history model: Second try

Key: CLIENT\_ID:YYYYMMDD\_HHMMSS

# Payments history model: Second try

Key: YYYYMM:CLIENT\_ID:DD\_HHMM\_SS

Get history for the client for current month is easy:

GETRANGE with prefix: 202212:1234:

Get history for two month is slightly less easier (2 reads)

GETRANGE with prefix: 202212:1234:

GETRANGE with prefix: 202211:1234:

How do we remove all data until 2022?

Just CLEAR RANGE 0, 2022



# How CLEAR RANGE works

Not like **DELETE FROM** table **WHERE** key **BETWEEN** a  
**AND** b

- **Effect** is immediate
  - Works through ‘applying’ this effect based on WAL
- **Actual disk work** is deferred until disk usage is low enough

# Transactional outbox: Definition

Pattern for transactionally changing data in DB  
and sending an event to Kafka/MQ/etc

# Transactional outbox: Naive solution

1. Insert ‘message’ object with other transactional changes
2. In other thread, fetch bunch of messages
3. Send messages the way you need
4. Delete message objects
5. Repeat

# Transactional outbox: Naive solution

1. Insert ‘message’ object with other transactional changes
2. In other thread, fetch bunch of messages
3. Send messages the way you need
- 4. Delete message objects: performance issue**
5. Repeat

# Transactional outbox: Better solution

We can eliminate atomic deletions if we would have monotonically increasing ID

Reader offset:  
3



ID	Message
1	{ ... }
2	{ ... }
3	{ ... }
4	{ ... }
5	{ ... }

# Transactional outbox: Better solution

We can eliminate atomic deletions if we would have monotonically increasing ID

Naive approach: lets use sequence

The problem is - we need for them to monotonically **appear** to other transactions

Step	Thread: Writer 1	Thread: Writer 2	Thread: Sender: Offset = 0
1	ID = 1 // Next ID <b>SET</b> 1, {}		
2		ID = 2 // Next ID <b>SET</b> 2, {}	
3		<b>COMMIT</b>	
4			ToSend = [ [2, {}] ] // GETRANGE 0, 255 Offset = 2
5	<b>COMMIT</b>		
6			ToSend = [] // GETRANGE 2, 255 Message with ID=1 is LOST

# Transactional outbox: Better solution

Oracle has SCN, Postgres - TXID - transaction sequence number: can not be used during commit :(

FoundationDB has Versionstamp - same as SCN/TXID  
Monotonically increasing 10 bytes number, ‘version’ of data in a whole cluster

MUTATE to the rescue: Replaces given part of key with Versionstamp **at commit time**

Step	Thread: Writer 1	Thread: Writer 2	Thread: Sender: Offset = 0
1	<b>MUTATE</b> 0000, {}		
2		<b>MUTATE</b> 0000, {}	
3		<b>COMMIT</b> DB Version: 0001 Inserted: 0001, {}	
4			ToSend = [ [1, {}] ] // GETRANGE 0, 255 Offset = 1
5	<b>COMMIT</b> DB Version: 0002 Inserted: 0002, {}		
6			ToSend = [ [2, {}] ] // GETRANGE 1, 255 Offset = 2

# How we implemented this layer

Originally - as a Kotlin coroutine-based library.

Library approach issues:

- Encryption/re-encryption + key management is easier in single place
- FoundationDB doesn't have notion of grants
- Harder troubleshooting (when need to dive into what's in DB)

Currently in process: library => separate service with transactional REST API



# Summary

Ordered Key-Value + Serializable isolation level is a powerful abstraction

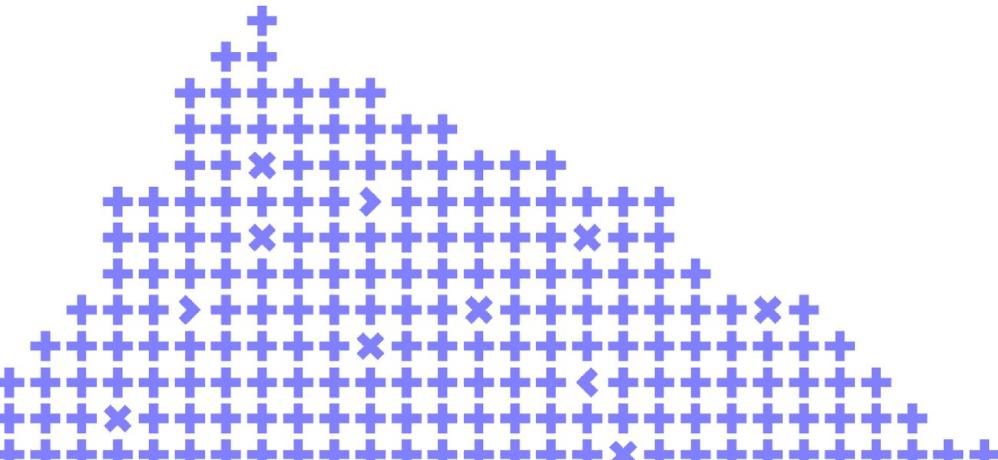
It's easy to implement higher level API yourself  
If RecordLayer fits you, don't roll your own

We already use it in production



# Leave your feedback!

You can rate the talk and  
give a feedback on what  
you've liked or what  
could be improved



Co-organizer

**Yandex**